Les systèmes d'exploitation

Wadoud Bousdira[1]
wadoud.bousdira@univ-orleans.fr

[1]LIFO, University of Orléans
Orléans, France

Orléans, 2023

Les processus légers (threads)

Définition

Un processus léger (thread) est un fil d'exécution qui partage l'espace d'adressage du processus, mais possède sa propre pile et ses valeurs de registres.

- Un processus léger partage les ressources avec les autres processus légers du même espace d'adressage ⇒
  -  facilite la communication entre les différents processus légers ;
  -  augmente le parallélisme du programme. (Ex. gestion d'une interface graphique).
- Le partage des ressources crée des problèmes de synchronisation pour garantir la cohérence et l'accès concurrent aux données.

Les processus légers

Il existe de nombreuses mises en œuvre différentes et plus ou moins efficaces des processus légers et de leur interaction avec le reste du système.

Section critique

Les processus légers accèdent en (pseudo-)parallèle aux mêmes données. Pour assurer la cohérence des données, certaines portions de code doivent être exécutées de manière atomique : ce sont les sections critiques.

Sections critiques

- Concept valant autant pour les processus légers que pour les processus accédant à des ressources en mémoire partagée.
- Très fréquentes dans les SE dont le rôle est le partage de ressources (critiques ou non) entre les processus.
- peuvent être emboîtées $\rightsquigarrow$ risque d'interblocage !

p_1	p_2
entrée SC_1	entrée SC_2
entrée SC_2	entrée SC_1
sortie SC_2	sortie SC_1
sortie SC_1	sortie SC_2

- Autre risque d'interblocage : deux processus ont besoin chacun de deux ressources, chacune d'elles est affectée à l'un d'entre eux.

Un aperçu du problème

une procédure qui affiche un caractère lu au clavier

```
void echo() {
    in = getchar();
    out = in;
    putchar(out);
}
```

Exécution invalide

2 processus p_1 et p_2 partageant l'application

1. p_1 appelle `echo`
2. p_1 est interrompu après l'écriture dans `in`
3. p_2 appelle `echo` et termine son appel
4. p_1 reprend la main, mauvaise valeur de `in`.

La source du problème

Le partage de ressources

On distingue trois degrés d'interaction

1. indépendance : processus indépendants, pas d'interaction ;
   Le SE doit gérer les accès concurrents aux ressources (périphériques,...)
2. interactions indirectes : les processus ne se connaissent pas mais accèdent à des données partagées ;
3. interactions directes : les processus se connaissent mutuellement et sont conçus pour concourir à la réalisation d'une tâche.

Dans les deux derniers cas, le SE doit permettre aux processus de coopérer.

Sections critiques

Mécanismes d'exclusion mutuelle

Mis en place pour assurer qu'au plus un processus (léger) accède à la section critique en même temps, quitte à *endormir* les processus (légers) en attente d'entrée.

Sections critiques

Le problème des variables partagées

Hypothèses d'exécution

- les évolutions de chaque processus (léger) sont a priori indépendantes ;
- le délai entre deux instructions d'un processus est non nul, mais fini ;
- deux accès à une case mémoire ne peuvent être simultanés ;
- les registres sont sauvegardés et restaurés à chaque commutation.

Sections critiques

être capable de coordonner les interactions entre activités simultanées pour éviter les conflits lecture/écriture.

Synchronisation

1. comment un processus fait-il pour passer de l'information à un autre processus ?
   Ex. pipeline du shell : `ls -l | wc -l`
   Facile pour les threads qui partagent l'espace d'adressage.
2. retarder l'exécution d'une activité dans l'attente de la terminaison d'une autre.
   Ex. deux processus de réservation tentent de récupérer pour deux clients le dernier siège disponible.
3. Séquençage en présence de dépendances.
   Ex. producteur/consommateur.

Sections critiques

Producteur/consommateur

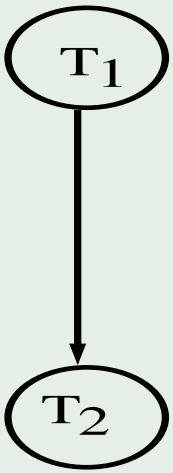
- processus `P` calcule une valeur $\rightsquigarrow$ producteur
- processus `Q` utilise la valeur calculée par `P` $\rightsquigarrow$ consommateur
- contrainte de synchronisation :

$$\text{date\_fin (produire(v))} \leq \text{date\_debut (consommer(a))}$$

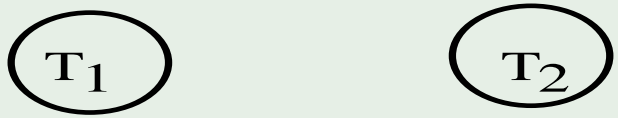
Représentation graphique de ces contraintes : graphe de précédence.

Graphe de précédence

Un nœud = une tâche ou une activité



$date\_fin(T_1) \leq date\_debut(T_2)$

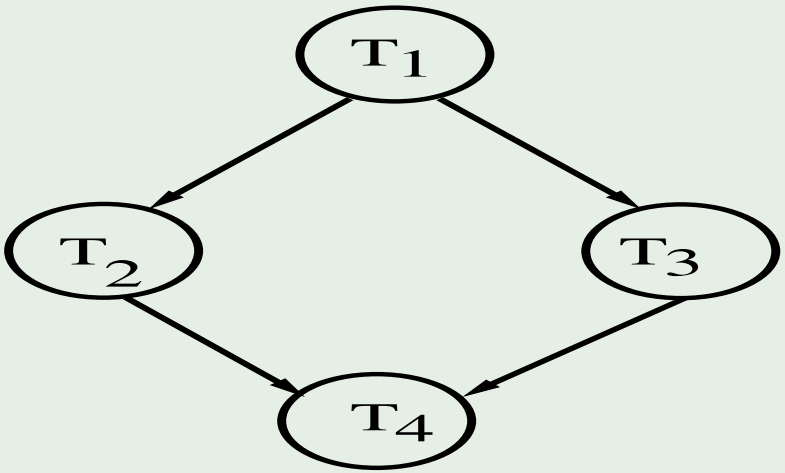


pas de contrainte

Graphe de précédence

Plusieurs déroulements possibles dans le temps :

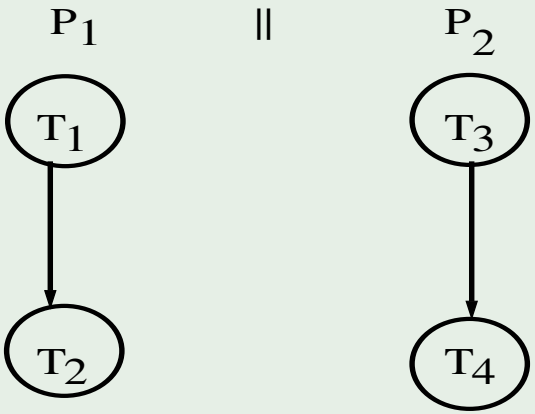
$$2calculs \begin{cases} T_1 & T_2 & T_3 & T_4 \\ T_1 & T_3 & T_2 & T_4 \end{cases}$$



Un déroulement possible = un calcul

Composition parallèle

Juxtaposition de deux graphes.



Notations :

$P_1 \parallel P_2$

parbegin

P_1 ; P_2

parend

6 calculs possibles de $P_1 \| P_2$: explosion combinatoire ?

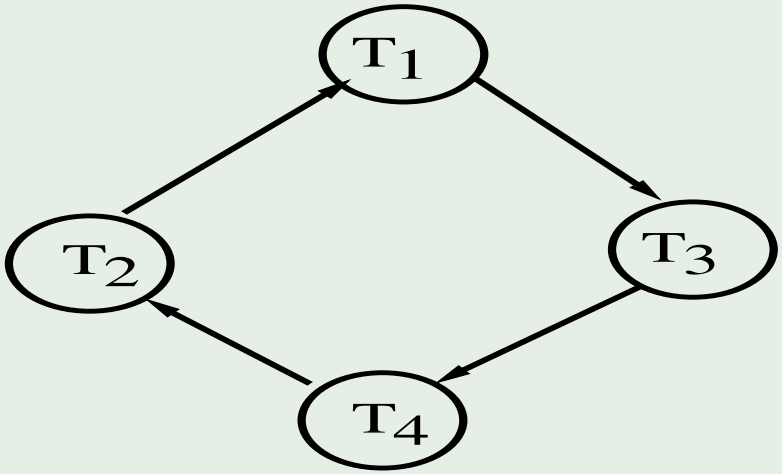
$$
\begin{array}{cccc}
T_1 & T_2 & T_3 & T_4 \\
T_1 & T_3 & T_2 & T_4 \\
T_1 & T_3 & T_4 & T_2 \\
T_3 & T_1 & T_2 & T_4 \\
T_3 & T_1 & T_4 & T_2 \\
T_3 & T_4 & T_1 & T_2
\end{array}
$$

Synchronisation

Attention ! un programme $\|$ peut ne pas terminer

- pour tous ces calculs
- pour seulement certains de ces calculs.

Blocage (Ex. carrefour) ou boucle infinie.



Synchronisation

Un programme $\|$ est correct

si tous les calculs terminent et donnent le résultat attendu.

- Aucun calcul ne conduit au blocage ;
- aucun calcul ne conduit à une boucle infinie ;
- tous les calculs donnent le même résultat.

Comment éviter d'examiner tous les calculs ?

Synchronisation

- Sur l'ensemble des instructions de programmes parallèles, peu d'entre elles peuvent provoquer des conflits d'entrelacement lors de l'exécution.
- ces zones particulières sont appelées sections critiques.

Synchronisation

Sections critiques

Ensemble de suites d'instructions pouvant produire des résultats imprévisibles lorsqu'elles sont exécutées simultanément par des processus différents.

$\rightsquigarrow$ Deux exécutions peuvent donner des résultats différents.

```
void echo() {
    in = getchar();
    out = in;
    putchar(out);
}
```

Synchronisation

Ensembles de sections critiques

- Une suite d'instructions sur des variables partagées est *éventuellement* une section critique relativement à d'autres suites d'instructions sur les mêmes variables partagées et non dans l'absolu.
- l'exécution simultanée de deux sections critiques appartenant à des ensembles différents et ne partageant pas de variable ne pose pas de problème.

Sections critiques

Problème : comment les déterminer ?

- l'existence de sections critiques implique l'utilisation de variables partagées, mais l'inverse n'est pas vrai.
- en pratique, les sections critiques doivent être détectées par les concepteurs des programmes et dès qu'il y a des variables partagées, il y a de fortes chances de se trouver en présence de sections critiques.

Sections critiques

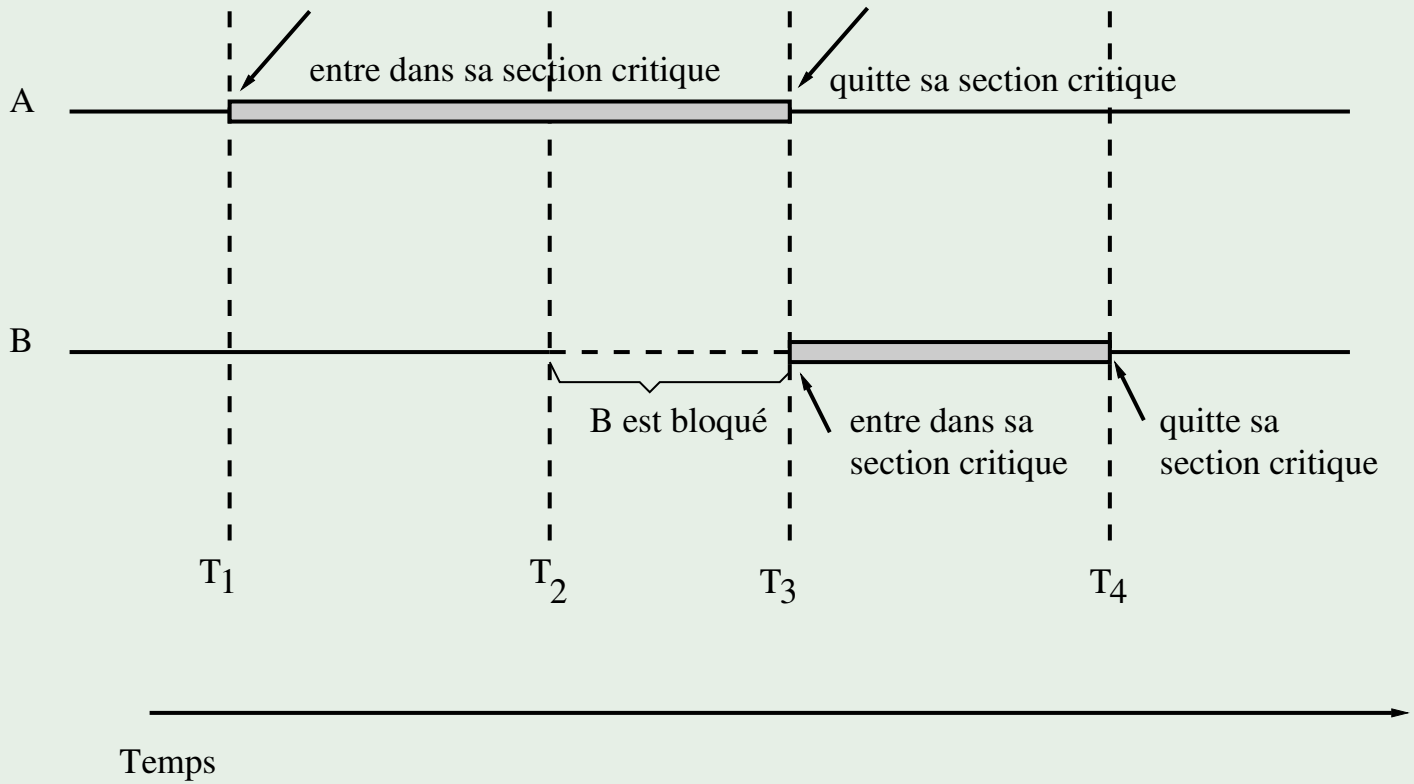
Vers une bonne coopération inter-processus

Quatre conditions à satisfaire :

1. Deux processus ne doivent pas se trouver simultanément dans leurs sections critiques.
2. Aucune hypothèse sur la vitesse ou le nombre de processus mis en œuvre.
3. Aucun processus s'exécutant à l'extérieur de sa section critique ne doit bloquer d'autres processus.
4. Aucun processus ne doit attendre indéfiniment pour pouvoir entrer dans sa section critique.

Sections critiques

Exemple de bon fonctionnement



Sections critiques

Exclusion mutuelle

- Avant d'entrer dans sa section critique, un processus doit s'assurer qu'aucun autre processus n'est en train d'exécuter une section critique du même ensemble.
- Dans le cas contraire, il ne devra pas progresser tant que l'autre processus n'aura pas terminé sa section critique.

Sections critiques

Exclusion mutuelle

1. Protocole d'entrée en section critique
   ensemble d'instructions qui permet cette vérification, et la non-progression éventuelle.
2. Protocole de sortie de section critique
   ensemble d'instructions qui permet à un processus ayant terminé sa section critique d'avertir les processus en attente que la voie est libre.

Sections critiques

Structure des processus avec sections critiques

Processus P_i :

```
début
    section non critique
    protocole d'entrée en section critique
    section critique
    protocole de sortie de section critique
fin
```

Sections critiques

Solutions de gestion des sections critiques

Désactiver les interruptions.

- Solution simple dans un système monoprocesseur.
- Chaque processus désactive les interruptions juste après son entrée en section critique, et les réactive juste après sa sortie.

Inconvénients :

- On donne le pouvoir à un processus utilisateur de désactiver les interruptions !
- Les E/S sont bloquées $\rightsquigarrow$ peut entraîner *le gel* du système ...

Sections critiques

Des solutions logicielles

- les sémaphores (E. W. Dijkstra)
- les moniteurs (C. A. Hoare)
- les bibliothèques de fonctions.

Acquérir un verrou, en mode maître.

Fonctionne parce que tous les processus en concurrence exécutent les mêmes appels système et obéissent aux mêmes conventions pour se synchroniser 

Solutions de gestion des sections critiques

Le verrou réduit à un bit

Variable unique partagée, valeur initiale = 0.
Processus P_i :

```
section non critique
tant que verrou != 0 faire rien ftq
verrou = 1
section critique
verrou = 0
section non critique
```

Le verrou réduit à un bit

Variable unique partagée, valeur initiale = 0.
Processus P_i :

```
section non critique
tant que verrou != 0 faire rien ftq
verrou = 1
section critique
verrou = 0
section non critique
```

Inconvénient : la consultation et la modification du verrou constituent elles-mêmes de nouvelles sections critiques !

Le verrou réduit à un bit

p0	p1
`lecture verrou (=0)`	`lecture verrou (=0)`
	`verrou = 1`
`verrou = 1`	

Entrée simultanée en section critique !

Solutions de gestion des sections critiques

Alternance stricte

Verrou + attente active : spin lock.

```
p0
tant que vrai faire
  tant que tour != 0 faire rien ftq
  section critique
  tour = 1
  section non critique
ftq
```

```
p1
tant que vrai faire
  tant que tour != 1 faire rien ftq
  section critique
  tour = 0
  section non critique
 ftq
```

Si p_0 bcp plus lent que p_1, p_1 peut se retrouver bloqué par p_0 hors de sa section critique.

Solutions de gestion des sections critiques

Algorithme de Peterson

Amélioration de l'algorithme de Dekker.

- une procédure pour entrer en section critique, et une procédure pour la quitter, développées en C ANSI.
- un tableau de booléens indicés par les numéros des processus, valeur = `true` pour ceux intéressés par l'entrée en SC.

Solutions de gestion des sections critiques

Algorithme de Peterson

Amélioration de l'algorithme de Dekker.

- une procédure pour entrer en section critique, et une procédure pour la quitter, développées en C ANSI.
- un tableau de booléens indicés par les numéros des processus, valeur = `true` pour ceux intéressés par l'entrée en SC.

```
Processus Pi
  section non critique
  entrer_SC(i)
  section critique
  quitter_SC(i)
  section non critique
```

```
entier tour booléen drapeau[N] : initialisé à false
```

Algorithme de Peterson

2 processus p_0, p_1

```
void entrer_SC(entier processus)
début
  entier other = 1 - processus
  drapeau[processus] = true
  tour = processus
  tant que tour = processus && drapeau[other] faire
    rien
  ftq
fin

void quitter_SC(entier processus)
début
  drapeau[processus] = false
fin
```

Algorithme de Peterson

Si p_0 et p_1 appellent simultanément `entrer_SC`,

- ils stockent leur numéro de processus dans `tour`.
- si `tour` est à 1, p_1 boucle et p_0 entre en section critique.
- p_1 boucle et n'entre pas en section critique tant que p_0 n'a pas quitté la sienne.

Algorithme de Peterson

Avec une aide matérielle

- `TSL RX, LOCK` (Test and Set Lock) : teste et positionne le verrou. Lit le contenu du mot mémoire `LOCK` dans le registre `RX` et stocke une valeur différente de 0 à l'adresse mémoire `LOCK` en deux opérations indivisibles (par verrouillage du bus mémoire).
- variable partagée `Lock`.
- si `Lock` est à 0, n'importe quel processus peut la positionner à 1 en utilisant `TSL`.

Algorithme de Peterson

Avec une aide matérielle

```
entrer_SC :
  TSL RX, LOCK
  CMP RX, #0
  JNE entrer_SC
  RET       // retourne à l'appelant, entre en SC

quitter_SC :
  MOVE LOCK, #0
  RET       // retourne à l'appelant
```

Solutions de gestion des sections critiques

Inconvénient des algorithmes précédents : l'attente active.

- perte de temps.
- peut provoquer un blocage définitif des processus.

Solutions de gestion des sections critiques

Problème du producteur/consommateur

Utilisation des primitives `sleep` et `wakeup`.

- Deux processus partagent un buffer commun de taille fixe,
- le producteur place des informations dans le buffer ;
  - il entre en sommeil lorsqu'il veut placer une information et que le buffer est plein,
  - il sera réveillé quand le consommateur aura supprimé un ou plusieurs éléments du tampon
- le consommateur récupère les informations du buffer ;
  - s'il veut récupérer un élément du tampon et que le buffer est vide, il entre en sommeil,
  - il sera réveillé quand le producteur aura déposé un ou plusieurs éléments dans le tampon.

Problème du producteur/consommateur

Une condition de concurrence.

Une variable `count` : nombre d'éléments dans le tampon.

`count` $\in$ `[0..N]` , `N` : capacité du buffer.

Code du producteur :

```
entier count=0
void producteur()
début
  entier item
  tant que true faire
    item=produire_item()
    si count=N alors sleep() fsi
    insérer_item(item)
    count=count + 1
    si count=1 alors wakeup(consommateur) fsi
  ftq
fin
```

Problème du producteur/consommateur

Code du consommateur :

```
void consommateur()
début
  entier item
  tant que true faire
    si count=0 alors sleep() fsi
    item=supprimer_item()
    count=count - 1
    si count=N-1 alors wakeup(producteur) fsi
    consommer_item(item)
  ftq
fin
```

Problème du producteur/consommateur

Condition de concurrence :

L'accès à `count` n'est pas contraint !

producteur	consommateur
	lit count. Buffer vide $\Rightarrow$ count $= 0$
	arrêt par l'ordonnanceur(ça peut arriver !)
insérer_item(item)	
count $\leftarrow 1$	
wakeup(consommateur)	
	teste `count` à 0 $\Rightarrow$ entre en sommeil
produit ...	
buffer plein $\Rightarrow$ entre en sommeil	

Problème : signal `wakeup` perdu car envoyé à un processus non dormant.

Gestion des sections critiques

Les sémaphores

Dus à Dijkstra en 1965, implantés pour la 1ère fois dans Algol 68.

Le principe

- Les processus coopèrent au moyen de signaux ;
- un processus peut être suspendu jusqu'à ce qu'il reçoive un signal émis par un autre processus.

Les sémaphores

Signaux transmis par des variables spéciales

Un sémaphore est une variable `S` de type entier à laquelle on ne peut accéder que par deux opérations :

1. `P` (puis-je ?)
2. `V` (vas-y !)

Les sémaphores

- `P` : je m'approprie le jeton si je peux
  entrée en section critique ?
- `V` : je restitue le jeton
  sortie de la section critique ?
- `S` : nombre de jetons.
  - si $S > 0$ la ressource est libre,
  - si $S \leq 0$, la ressource est occupée, et $|S|$ vaut le nombre de processus attendant pour y accéder.

Noyau Linux : `down` (P), `up` (V).

Les sémaphores

Opérations atomiques, appels système

```
procédure P(S)
début
   tant que S <= 0 faire rien ftq
   S = S-1
fin
```

```
procédure V(S)
début
   S = S+1
fin
```

Les sémaphores

Opérations atomiques, appels système

parfois,

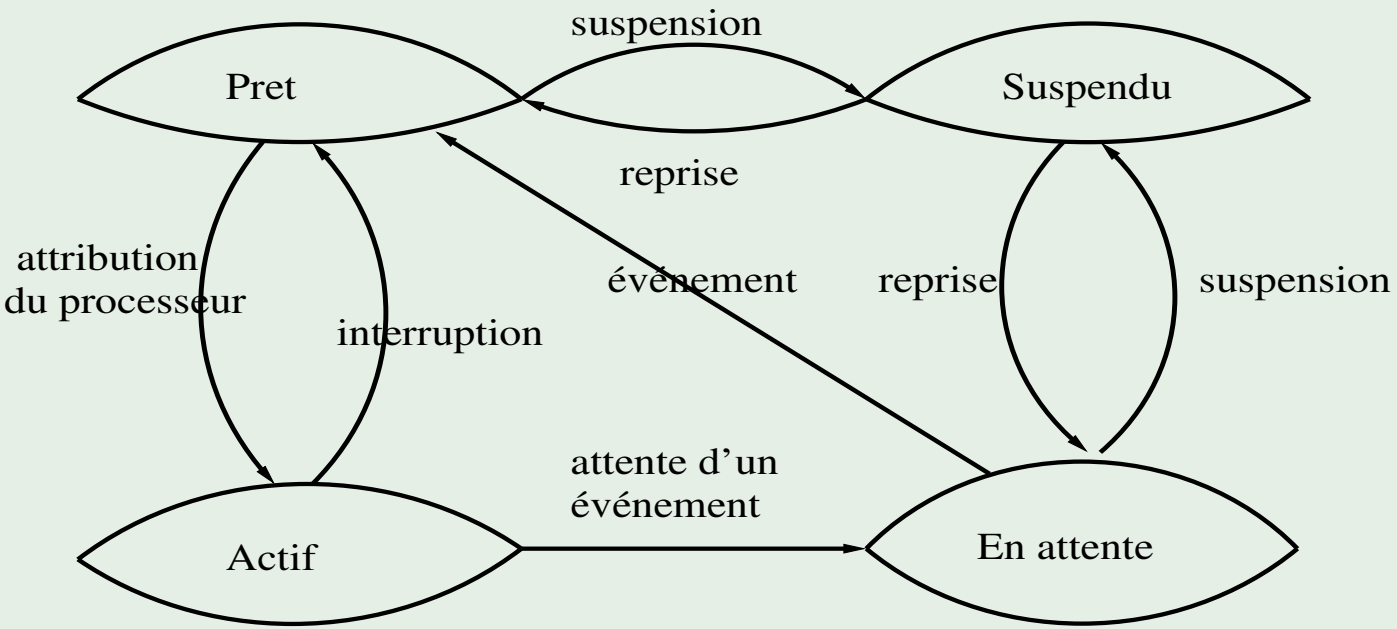
```
procédure init(S, valeur)
début
    S = valeur
fin
```

permet une seule fois d'initialiser la valeur du sémaphore.

Les sémaphores

- Le SE désactive toutes les interruptions pendant un très court laps de temps, pour tester le sémaphore, l'actualiser et placer si nécessaire le processus concerné en sommeil.
- Si plusieurs processeurs sont utilisés, chaque sémaphore doit être protégé par une variable verrou, avec l'instruction `TSL`, pour garantir qu'un seul processus à la fois teste la valeur du sémaphore.
- L'accès protégé du sémaphore ne prend que quelques microsecondes $\neq$ attente active des processus.

Un nouvel état pour les processus



Liste des processus en attente, gérée par l'ordonnanceur.

Les sémaphores

Les algorithmes réalisant une exclusion mutuelle sont difficiles à généraliser.

Sémaphore : types de données abstrait composé

- d'un compteur entier, initialisé à une valeur constante ;
- de l'opération `P` ;
- de l'opération `V`.

Les sémaphores

```
procédure P(sema : sémaphore)
    tant que sema.compteur=0 faire
        suspendre(sema)
    ftq
    sema.compteur=sema.compteur - 1
fin
```

```
procédure V(sema : sémaphore)
    sema.compteur=sema.compteur + 1
    réveiller(sema)
fin
```

Les sémaphores binaires

Définition

- Compteur initialisé à 1.
- Utilisé par deux processus ou plus pour faire en sorte que seul l'un d'entre eux pourra entrer en section critique à un instant donné.
- Si chaque processus effectue un `P` avant d'entrer en section critique, et un `V` juste avant de la quitter, l'exclusion mutuelle est garantie.

Les sémaphores

Réalisation

Structure système :

```
sémaphore =
   structure
      compteur : entier
      file : liste [d'identificateurs] de processus
   fin de structure
```

- `compteur` contient initialement une valeur `val`. Appelé *niveau du sémaphore*.
- L'identificateur de processus permet de retrouver le bloc de contrôle du processus (i.e. son état, son numéro, ses ressources, sa localisation en mémoire, le contenu de ses registres etc...)

Les sémaphores

Réalisation logicielle

Traduction directe de la spécification fonctionnelle.

L'opération `init`

```
procédure init(sem : sémaphore, entier val)
début
    masquer_it
    sem.compteur=val
    sem.file=vide
    démasquer_it
fin
```

Les sémaphores

```
procédure P(sem : sémaphore)
début
   masquer_it
   si sem.compteur > 0 alors
      sem.compteur=sem.compteur - 1
   sinon
      suspendre le processus demandeur
      mettre son identificateur dans sem.file
      bloquer ce processus
   fsi
   démasquer_it
fin
```

Les sémaphores

```
procédure V(sema : sémaphore)
   masquer_it
   si vide(sema.file) alors
      sema.compteur = sema.compteur + 1
   sinon
      enlever un identificateur de sem.file
      réveiller le processus correspondant
      (il passe à l'état prêt)
   fsi
   démasquer_it
fin
```

Réalisation logicielle

Remarques

- On suppose que `suspendre` et `réveiller` n'ont pas d'identificateur d'attente.
- Par rapport à la spécification fonctionnelle, c'est le processus exécutant un `V` qui réévalue la condition permettant de réveiller un processus suspendu. Celui-ci pourra ensuite continuer son exécution dès qu'il aura reçu un processeur, comme s'il n'avait pas été suspendu.
- La politique de gestion de la file peut être quelconque (priorité, LIFO, FIFO, ...). Si elle n'est pas FIFO, il y a risque de famine.
- Si un `V` débloque un processus, soit le processeur est alloué au processus débloqué, soit il est laissé au processus qui a exécuté le `V` (cas le plus fréquent).

Les sémaphores

Réalisation matérielle

- De nombreuses machines proposent maintenant des opérations `P` et `V` microprogrammées ou câblées,
  $\rightsquigarrow$ règle aussi le problème de l'exclusion mutuelle pour l'accès au compteur et à la file.
- Réalisation dans un système réparti sans mémoire commune : la gestion des sémaphores est généralement confiée à un processeur spécifique.

Les sémaphores

‘

Les `mutex` : réalisation d'exclusion mutuelle par des sémaphores

- Version simplifiée des sémaphores, quand les décomptes ne sont pas nécessaires.
- `mutex` est une variable $\in \{0,\ 1\}$. 1=`déverouillé`, 0=`vérouillé`.
- prélude : `P(mutex)`, postlude : `V(mutex)`.

processus p_1	processus p_2
P(mutex)	P(mutex)
section critique	section critique
V(mutex)	V(mutex)

- Pas deux processus en section critique à la fois
- lorsqu'un processus quitte la section critique, il rétablit `mutex` à 1, ce qui permet à un autre éventuel processus bloqué d'entrer en section critique.

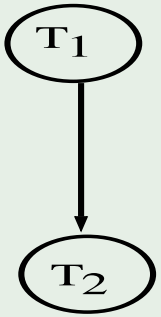
Les sémaphores

```
init(mutex, 1)
Réservation
P(mutex)          // prélude de SC
si nbplace > 0 alors
   réserver une place
   nbplace=nbplace - 1
fsi
V(mutex)       // postlude de SC
```

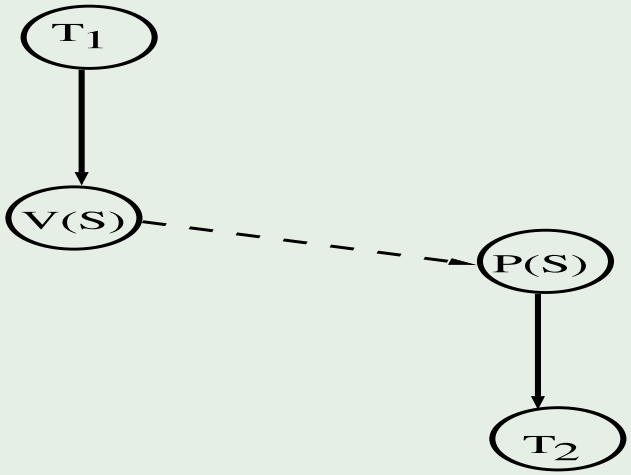
‘

Utilisation des sémaphores dans les graphes de précédence

Deux tâches T_1 et T_2 tq $T_1 < T_2$

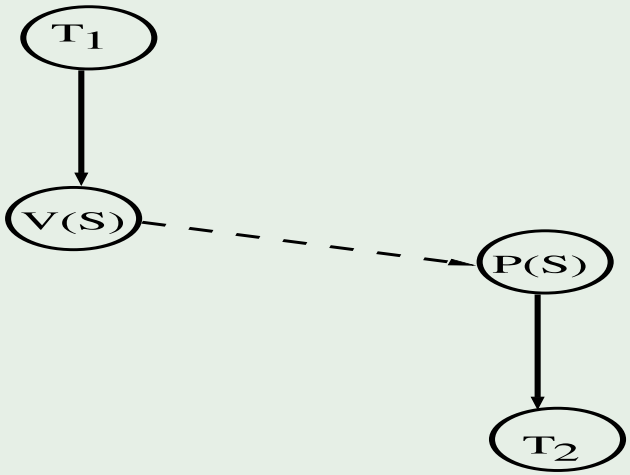


1. `début` T_1 `;` T_2 `fin`
2. si T_1 et T_2 sont des tâches de 2 processus différents qui doivent être synchronisées :



Utilisation des sémaphores dans les graphes de précédence

‘



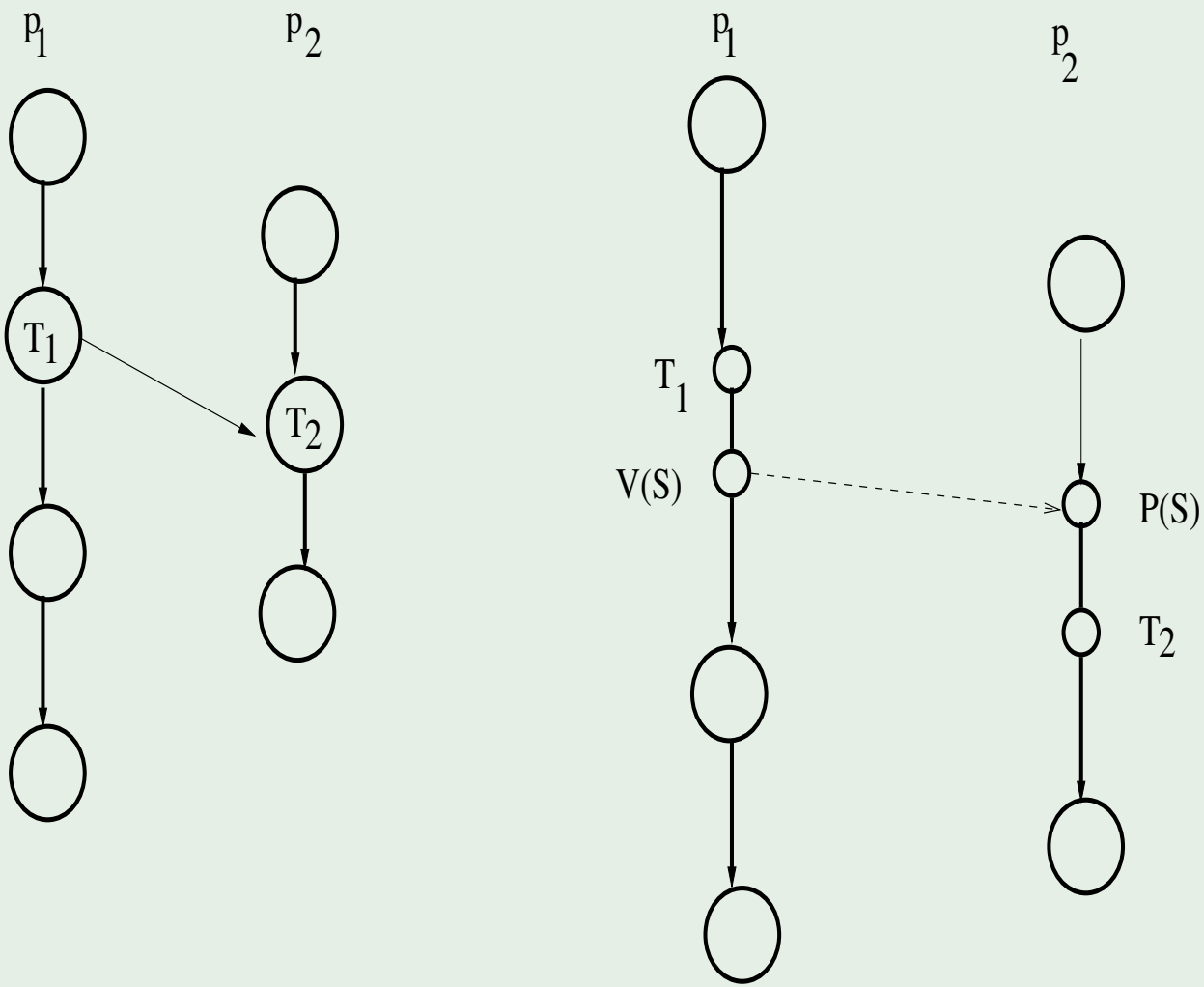
La valeur initiale de S $= 0 \Rightarrow$ `P(S)` ne peut s'exécuter que si `V(S)` l'a été.

Force la tâche T_1 à précéder la tâche T_2.

Utilisation des sémaphores dans les graphes de précédence

‘

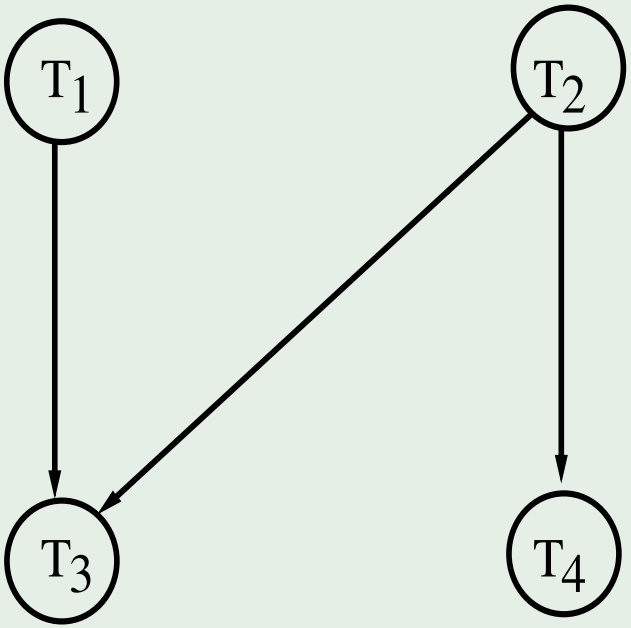
Avec 2 processus différents :



Utilisation des sémaphores dans les graphes de précédence

‘

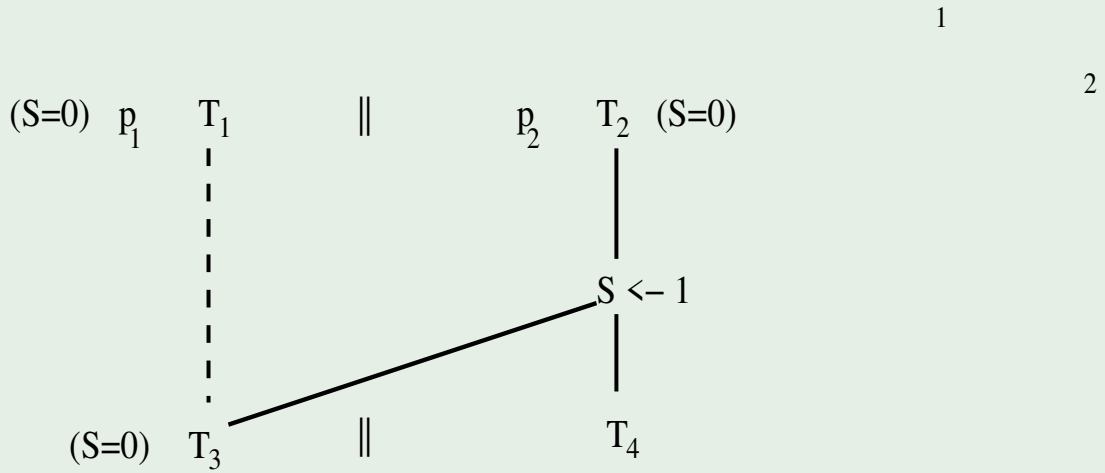
Exemple



Utilisation des sémaphores dans les graphes de précédence

Exemple 1

- ```
  init(S,0);
  parbegin
      début T1; P(S); T3 fin
      début T2; V(S); T4 fin
  parend
  ```


  ```

Utilisation des sémaphores dans les graphes de précédence

Exemple 1

Exécuter les 4 tâches en //, exprimer les relations de précédence par des sémaphores :

```
init(S,0); init(R,0); init(U,0)
parbegin
    début T1; V(S) fin
    début T2; V(R); V(U) fin
    début P(R); P(S); T3 fin
    début P(U); T4 fin
parend
```

Utilisation des sémaphores dans les graphes de précédence

Exemple 2

k exemplaires d'une ressource, $k \geq 2$ $\Rightarrow$ jusqu'à au plus k processus en section critique.

- Sémaphore `nb_ressources` qq

  `init(nb_ressources, k)`

- chaque processus exécute :

```
répéter
    section non critique
    P(nb_ressources)
    section critique
    V(nb_ressources)
jusqu'à faux
```

Les sémaphores

Inconvénients

Les opérations sont à la charge du programmeur 

- s'assurer que les opérations `P` et `V` apparaissent dans le bon ordre et se correspondent bien ;
- vérifier que les sections critiques disséminées dans le code sont convenablement protégées.

Une autre approche pour résoudre les problèmes de synchronisation : les moniteurs.

Les moniteurs

- Primitive de synchronisation de haut niveau due à Hoare et Brinch Hansen en 1973.
- Dans certains langages de programmation, ex. Concurrent Pascal, le moniteur est défini comme un type abstrait de données, dans la partie déclaration du programme. Structure de procédure sans paramètre
  - déclaration des variables partagées, non accessibles en dehors du moniteur ;
  - procédures et fonctions internes au moniteur, les seules à manipuler les variables partagées ;
  - un corps, comportant l'initialisation des variables partagées.

Les moniteurs

Exemple : les gardiens compteurs

2 gardiens situés aux 2 entrées d'un magasin, comptent le nombre de clients de la journée en incrémentant au fur et à mesure, une variable commune N.

```
type nb_clients=moniteur
  var N : entier
  procédure incrementer;
  début
     N=N+1
  fin
  début
     N=0
  fin
```

Les moniteurs

Exemple : les gardiens compteurs

Un même code pour les processus p_1 et p_2 :

```
début
   tant que magasin_ouvert faire
      si entrée alors nb_clients.incrementer
      fsi
   ftq
fin
```

Les moniteurs

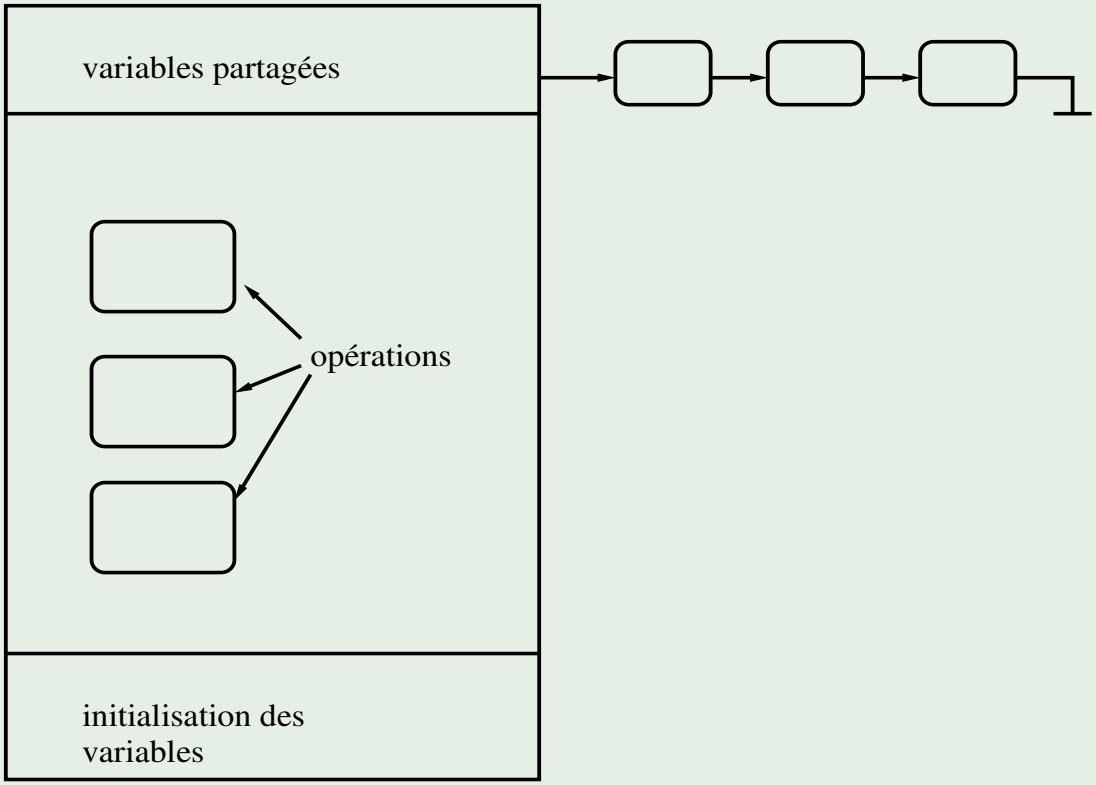
Avantages

- Les sections critiques sont transformées en fonctions ou procédures d'un moniteur. Elles ne sont pas dispersées.
  Chaque exécution de ces fonctions ou procédures est une opération atomique.
- La gestion des sections critiques n'est plus à la charge de l'utilisateur. Elle est réalisée par l'implantation du moniteur.

Le moniteur tout entier est implanté comme une section critique.

Les moniteurs

Schématiquement



Si le moniteur est occupé, le bloc de contrôle du processus est placé dans une file d'attente associée.

Les moniteurs avec condition

Variable de type condition d'un moniteur

Un type `condition`.

`X : condition`. `X` désigne une file de processus en attente, utilisée uniquement via les opérations `wait` et `signal`.

- si `p` exécute `X.wait` : `p` est mis en attente dans la file `X`. L'accès au moniteur redevient possible pour un autre processus.
- si `p` exécute `X.signal` : déclenche le réveil d'un autre processus en attente dans la file `X`. Si la file est vide, pas d'effet.
  1. choix du processus à réveiller ;
  2. choix du processus qui continue à être exécuté à l'intérieur du moniteur : (`p` ou celui qui a été réveillé (`q`) ?)

Les moniteurs

1. Processus à réveiller : gestion d'une FIFO.
2. Pas d'argument pour une solution précise.
   - Hoare : faire attendre `p` jusqu'à ce que le processus `q` quitte le moniteur ou soit mis en attente par une autre condition.
   - Concurrent Pascal : `p` quitte immédiatement le moniteur.

Les moniteurs avec condition

Exemple 1

n processus avec accès en exclusion mutuelle à un exemplaire unique d'une ressource $\rightsquigarrow$ gérer la ressource par un moniteur :

```
type allocateur=moniteur
   var occupé : booléen
   libre : condition
   procédure acquisition
   début
      si occupé alors libre.wait fsi
      occupé=true
   fin
```

Les moniteurs avec condition

```
procédure libération
début
   occupé=false
   libre.signal
fin
début
   occupé=false
fin
```

Analogie des procédures du moniteur avec les opérations `P` et `V` des sémaphores.

Les moniteurs avec condition

Exemple 2 : le Rendez-vous

`N` processus se donnent RV en un point donné : avant de continuer leurs exécutions, ils doivent attendre que tous soient arrivés au point de RV.

- Quand un processus arrive au point de RV, il s'enregistre (`n=n+1`)
- tant que `n` $<$ `N` attendre

Exemple 2

```
type RV=moniteur
  var n : entier
  touslà : condition
  procédure arrive
  début
     n=n+1
     si n < N alors touslà.wait sinon touslà.signal fsi
  fin
début
  n=0
fin
```

Exemple 3

Problème du producteur/consommateur

```
type tampon_borné = moniteur
   var tampon: tableau[0 .. n-1] de objet
   entrée, sortie : 0..n-1
   compte : 0..n
   de_la_place, des_objets : condition
```

Problème du producteur/consommateur

```
procédure ajouter(objetpro : objet)
début
   si compte=n alors de_la_place.wait fsi
   tampon[entrée] = objetpro
   compte=compte + 1
   entrée=(entrée + 1) mod n
   des_objets.signal
fin
```

Problème du producteur/consommateur

```
procédure prendre(var objetconbs : objet)
début
   si compte=0 alors des_objets.wait fsi
   objetcons=tampon[sortie]
   compte=compte - 1
   sortie=(sortie + 1) mod n
   de_la_place.signal
fin
```

Problème du producteur/consommateur

```
début      // du moniteur
   entrée=0
   sortie=0
   compte=0
fin
```

Problème du producteur/consommateur

```
procédure producteur
    var objetpro : objet
    début
    répéter
        produire(objetpro)
        ajouter(objetpro)
    jusqu'à false
fin
```

Problème du producteur/consommateur

```
procédure consommateur
   var objetcons : objet
   début
   répéter
      prendre(objetcons)
      consommer(objetcons)
   jusqu'à false
fin
```

Problème du producteur/consommateur

```
début // du pg principal
   parbegin
      producteur
      consommateur
   parend
fin
```

La synchronisation en Java

Le modèle

- plusieurs *fils* d'exécution appelés threads.
  Chaque thread a sa propre pile.
- un ordonnanceur qui demande l'exécution des threads sur les cœurs physiques, pendant un quantum de temps.
- une seule mémoire partagée appelée le tas
  - Unified Memory Access ou NUMA
  - Utilisée pour l'échange d'information entre les threads.

La synchronisation en Java

La synchronisation entre `threads` utilise des méthodes de la classe `Object`.
À l'intérieur d'un objet, un `thread` utilise :

- `wait()` met le `thread` en attente et relâche l'accès à l'objet
- `notify()` réveille un des processus qui attendent l'objet par `wait()` le choix est arbitraire et n'est pas obligatoirement FIFO
- `notifyAll()` réveille tous les processus attendant l'objet par `wait()`, ils seront exécutés dans un ordre quelconque.

doivent être lancées dans des méthodes synchronized.

La synchronisation en Java

Deux façons de créer des threads en Java :

1. créer une instance d'une classe *fille* de la classe `Thread` ; la classe fille doit redéfinir la méthode `run()`
2. utiliser le constructeur `Thread(Runnable)` de la classe `Thread` :
   - créer un `Runnable` (le code qui sera contrôlé par le contrôleur)
   - le passer au constructeur de `Thread`.

La synchronisation en Java

Lancement des threads

- On appelle la méthode `start()` du contrôleur de thread.
- Le code du `Runnable` s'exécute en parallèle au code qui a lancé le thread.

Ne pas appeler directement la méthode `run()` ! elle serait exécutée par le thread qui l'a appelée et non par un nouveau thread.

La synchronisation en Java

- En java, chaque moniteur n'a qu'une variable condition anonyme. Toutes les opérations `wait, notify, notifyAll` concernent cette condition anonyme.
- Le moniteur en Java utilise la sémantique de réveil *signal and continue* : le processus qui exécute `notify` ou `notifyAll` garde le moniteur et continue à s'exécuter dans l'objet.
- Un processus qui invoque le moniteur peut y avoir accès avant un processus réveillé par une opération `notify`.

Implantation en Java du producteur/Consommateur

```
public class Buffer {
   protected final int N;
   protected final Object[] buffer ;
   protected int count=0, nextIn=0; nextOut=0 ;

  public Buffer(int N) {
     this.N = N ;
     this.buffer = new Object[N];
   }

   public synchronized void insert(Object item) {
      while (count == N) {
         System.out.println("dort sur insert ");
         try {
             wait() ;
         } catch (InterruptedException e) { }
        System.out.println("réveil sur insert ");
      }
      buffer[nextIn] = item ;   //dépose une valeur dans le tampon
      nextIn = (nextIn+1) %N ;
      count++ ;
      notify() ; // réveille un processus en attente s'il y en a
    }
```

Implantation en Java du producteur/Consommateur

```
  public synchronized Object remove() {
     Object result ;
     while (count == 0)  {
        System.out.println("dort sur remove ");
        try {
           wait() ;
        } catch (InterruptedException e) { }
        System.out.println("réveil sur remove ");
       }
      result= buffer[nextOut] ;  // prélève une valeur du tampon
      nextOut = (nextOut+1) % N ;
      count--;
  notify(); // réveille un processus en attente s'il y en a
  return result;
  }

}  // Fin Buffer
```

Implantation en Java du producteur/Consommateur

```
public class Producteur extends Thread { // type processus producteur
   protected final Buffer buffer;

   public Producteur(Buffer buffer) {
      this.buffer=buffer;
   }

   public void run() {
      for (int i = 0; i < 100; i++) {
         buffer.insert(new Integer(i)); // crée un objet de type Integer
         try {
            sleep( (int)(Math.random() * 1000) );
         } catch (InterruptedException e) { } //en réponse à l'exception
      }
    }
}   // Producteur
```

```
public class Consommateur extends Thread {  // type processus consommateur
   protected final Buffer buffer;

   public Consommateur(Buffer buffer) {
      this.buffer=buffer;
   }

   public void run() {
      int k=0;
      try {
         for (int j=1; j<100; j++)  {
           Integer p = (Integer) buffer.remove();
           k = p.intValue();
           Thread.sleep( (int)(Math.random() * 100) );
         }  // fin du bloc où est levée l'exception
       }
     catch (InterruptedException e) { } // en réponse à l'exception
   }
}   // fin de Consommateur
```

```
public class ProducteurConsommateur { // objet programme principal
   static Buffer buffer = new Buffer(20); // tampon commun

   public static void main (String[] args) {
     Producteur p = new Producteur(buffer);
     Consommateur c = new Consommateur(buffer); // on passe l'objet partagé
     p.start() ;
     c.start();
    }  // fin de la méthode main

}   // fin de la classe ProducteurConsommateur
```

Exemple 4

Problème des lecteurs/rédacteurs

- un objet partagé par plusieurs processus ;
- l'objet n'est accessible que par deux opérations distinctes : lecture et écriture ;
- plusieurs lectures possibles en même temps ;
- une seule écriture en même temps : aucune autre écriture, ni aucune autre lecture.

Problème des lecteurs/rédacteurs

2 catégories de processus : les lecteurs et les rédacteurs.

Plusieurs variantes :

- traitement des processus dans l'ordre d'arrivée, avec exécution des lecteurs en parallèle.
- donner la priorité aux lecteurs $\rightsquigarrow$ peut provoquer un phénomène de famine.

Problème des lecteurs/rédacteurs

une solution fondée sur les règles :

- lorsque des lectures sont en attente de la terminaison d'une écriture, elles sont prioritaires sur la prochaine écriture ;
- lorsque des écritures sont en attente, une nouvelle lecture est placée en attente de la terminaison de l'écriture en cours.

distinguer

- un processus qui effectue une nouvelle demande pour une opération de lecture ou d'écriture
- d'un processus en attente ayant déjà fait cette demande.

Structure du moniteur

```
type lire_écrire=moniteur
  écriture : booléen;
  lecteurs : entier;
  accord_lecture, accord_écriture : condition;
```

La lecture

```
procédure début_lecture
début
  si écriture ou accord_écriture.non_vide alors
     accord_lecture.wait
  fsi
  lecteurs=lecteurs + 1 ;
  accord_lecture.signal
 fin;

 procédure fin_lecture
 début
   lecteurs=lecteurs - 1 ;
   si lecteurs=0 alors accord_écriture.signal fsi ;
 fin;
```

L'écriture

```
procédure début_écriture
début
   si lecteurs > 0 ou écriture alors accord_écriture.wait
   fsi
   écriture=true  ;
fin;

procédure fin_écriture
début
   écriture=false ;
   si accord_lecture.non_vide alors accord_lecture.signal
     sinon accord_écriture.signal
   fsi ;
fin;
```

Initialisation

```
début
    écriture=false;
    lecteurs=0;
fin
```

Problème des lecteurs/rédacteurs

	L_1	L_2	E_1	L_3	E_2	L_4	L_5
temps exécution	2	4	2	2	3	2	3
temps arrivée	0	1	3	4	7	8	9

t=0	L_1, lecteurs=1
t=1	$L_1 \parallel L_2$, lecteurs=2
t=2	L_1 a fini, L_2, lecteurs=1,
t=3	L_2, lecteurs=1, E_1 attend
t=4	L_2, lecteurs=1, E_1 attend, L_3 attend
t=5	L_2 a fini, lecteurs=0, E_1,ecriture=true, L_3 attend
t=7	E_1 a fini, ecriture=false, L_3, lecteurs=1, E_2 attend
t=8	L_3, lecteurs=1, E_2 attend, L_4 attend
t=9	L_3 a fini, lecteurs=0, E_2, ecriture=true, L_4, L_5 attendent
t=12	E_2 a fini, ecriture=false, $L_4 \parallel L_5$, lecteurs=2
t=14	L_4 a fini, L_5, lecteurs=1,
t=15	L_5 a fini, lecteurs=0

Problème des lecteurs/rédacteurs

- Tout opération de lecture (écriture) doit être de la forme :

```
début_lecture
< lecture >
fin_lecture
```

```
début_écriture
< écriture >
fin_écriture
```

- À la fin d'une opération d'écriture, réveil en cascade des lecteurs en attente. Pendant toute la durée de ces opérations, un processus quelconque, lecteur ou rédacteur, est bloqué à l'entrée du moniteur.

Illustration

La boîte aux lettres

- de type producteur-consommateur
- Un ensemble de personnes dispose d'une seule boîte aux lettres. Particularité : ne peut contenir qu'une seule lettre.
- un producteur peut y déposer une lettre $\rightsquigarrow$ boîte pleine ; un consommateur peut en retirer une lettre $\rightsquigarrow$ boîte vide.
- Les lettres ont un auteur mais pas de destinataire particulier, n'importe quel consommateur peut retirer une lettre.
- Chaque personne est un thread.

Illustration

Le programme :

- Deux producteurs et deux consommateurs.
- Absence d'interblocage :
  - si un thread est en attente, bloqué par une instruction `wait`, pour déposer (resp. retirer) une lettre, c'est que la boîte est pleine (resp. vide) ;
    on utilise instruction `notifyAll`. Il n'y a donc pas à la fois un consommateur et un producteur qui attendent.
  - Le nombre total de lettres déposées est prévu pour être égal au nombre total de lettres retirées. Si un consommateur (resp. un producteur) attend, c'est qu'il y a encore au moins une lettre qui sera déposée (resp. retirée). Par ailleurs, aucun producteur (resp. consommateur) n'est bloqué.

Les moniteurs

- Ils sont un concept du langage. Le compilateur doit les reconnaître et prendre en charge l'exclusion mutuelle.
- Existent en Java, mais pas dans beaucoup de langages (ex. C).
- Sémaphores et moniteurs inapplicables dans un système distribué composé de plusieurs processeurs, chacun ayant sa propre mémoire, et connecté en réseau local.

La synchronisation

L'échange de messages

MPI (Message Passing Interface) : standard de communication pour des nœuds exécutant des programmes parallèles sur des systèmes à mémoire distribuée.
Bonnes performances sur des machines massivement parallèles à mémoire partagée et sur des clusters d'ordinateurs hétérogènes à mémoire distribuée.

- Méthode de communication qui utilise deux primitives qui sont des appels système :
  1. `send(destination, &message)`
  2. `receive(source, &message)`
  - Si pas de message, le récepteur se bloque jusqu'à ce qu'il arrive, ou il retourne immédiatement un message d'erreur
  - pour pallier la perte de message, le récepteur envoie un accusé de réception à l'émetteur. Passé un certain délai, si l'émetteur n'a pas reçu d'accusé de réception, il renvoie de nouveau le message.

L'échange de messages

- Problème d'authentification : le client doit s'assurer qu'il communique avec le véritable serveur de fichiers, et non avec un imposteur.
- Si l'émetteur et le récepteur se trouvent sur le même ordinateur, l'échange de message est plus lent que le recours aux sémaphores ou aux moniteurs.

BSP

Les barrières

- vise les groupes de processus.
- Applications décomposées en phases :
  - phase de calcul, chaque processus effectue son calcul indépendamment des autres ;
  - phase de synchronisation : chaque processus atteignant la barrière attend que tous les autres l'atteignent.
  - Une fois que tous les processus ont atteint la barrière, phase de synchronisation (communication des processus entre eux).

Les barrières

BSP (Bulk Synchronous Parallel)